AMD
Smarter Choice

Ultimate Graphics Performance for DirectX 10 Hardware

Nicolas Thibieroz
European Developer Relations
AMD Graphics Products Group
nicolas.thibieroz@amd.com

V1.01

Generic API Usage

AMD Smarter Choice

DX10 designed for performance

- No legacy code
- No fixed function
- Validation at creation time
- Immutable state objects
- User mode driver
- Powerful API

Microsoft® DirectX® 10

State Management

AMD
Smarter Choice

DX10 uses immutable “state objects”

DX10 state objects require a new way to manage states

- A naïve DX9 to DX10 port ***will*** cause problems here
- Always create state objects at load-time
- Avoid duplicating state objects
- Recommendation to sort by states still valid in DX10!

Constant Buffers Management

Major cause of slow performance in current DX10 apps!

Constants are declared in buffers in DX10

```
cbuffer PerFrameUpdateConstants
{
    float4x4 mView;
    float    fTime;
    float3   fWindForce;
    // etc.
};
```

```
cbuffer SkinningMatricesConstants
{
    float4x4 mSkin[64];
};
```

When a CB has been updated and bound to a shader stage its ***whole*** contents are uploaded to the GPU

Need to strike a good balance between:

- Amount of constant data to upload
- Number of calls required to do it

Constant Buffers Management (2)

Always use a pool of constant buffers ***sorted by frequency of updates***

- Don't go overboard with number of CBs!
- Less than 5 is a good target
- CB sharing between shader stages can be a good thing

Global constant buffer unlikely to yield good performance

- Especially with regard to CB contention

Group constants by access patterns in a given buffer

```
cbuffer PerFrameConstants
{
   float4   vLightVector;
   float4   vLightColor;
   float4   vOtherStuff[32];
};
```

GOOD

```
cbuffer PerFrameConstants
{
   float4   vLightVector;
   float4   vOtherStuff[32];
   float4   vLightColor;
};
```

BAD

Resource Updates

AMD Smarter Choice

In-game creation and destruction of resources is slow!

- Runtime validation, driver checks, memory allocation...

Take into account for resource management

- Especially with regard to texture management

Create all resources in non-performance situations

- Up-front, level load, cutscenes, etc.

At run-time *replace* contents of existing resources rather than destroying/creating new ones

Resource Updates: Textures

AMD Smarter Choice

Avoid **`UpdateSubresource()`** for texture updates

- Slow path in DX10 (think **`DrawPrimitiveUP()`** in DX9)
- Especially bad with larger textures!
- E.g. texture atlas, imposter pages, streaming data...

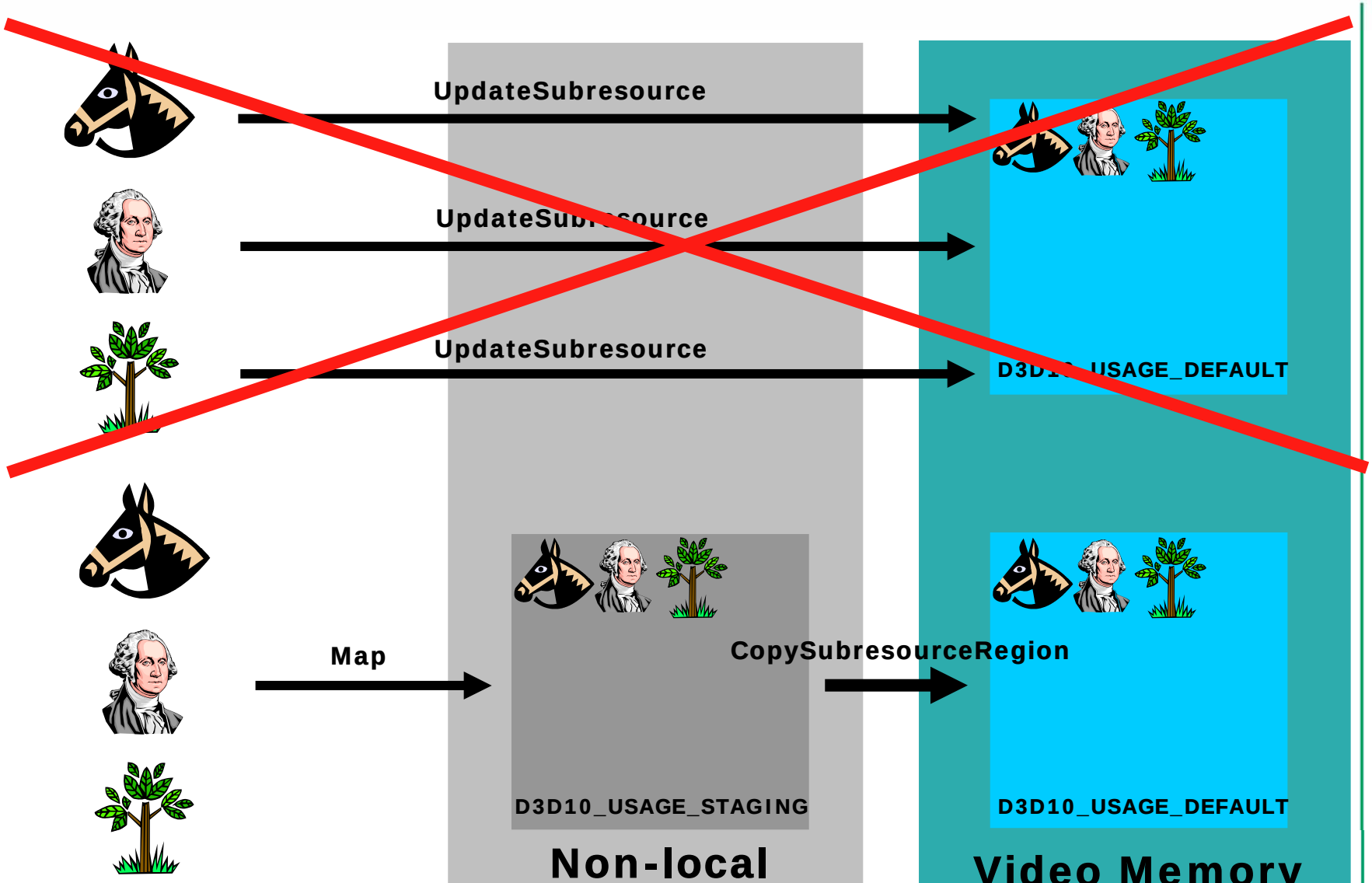
Perform all updates into pool of **`D3D10_USAGE_STAGING`** textures

- Use **`Map(D3D10_MAP_WRITE, ...)`** with **`D3D10_MAP_FLAG_DO_NOT_WAIT`** to avoid stalls

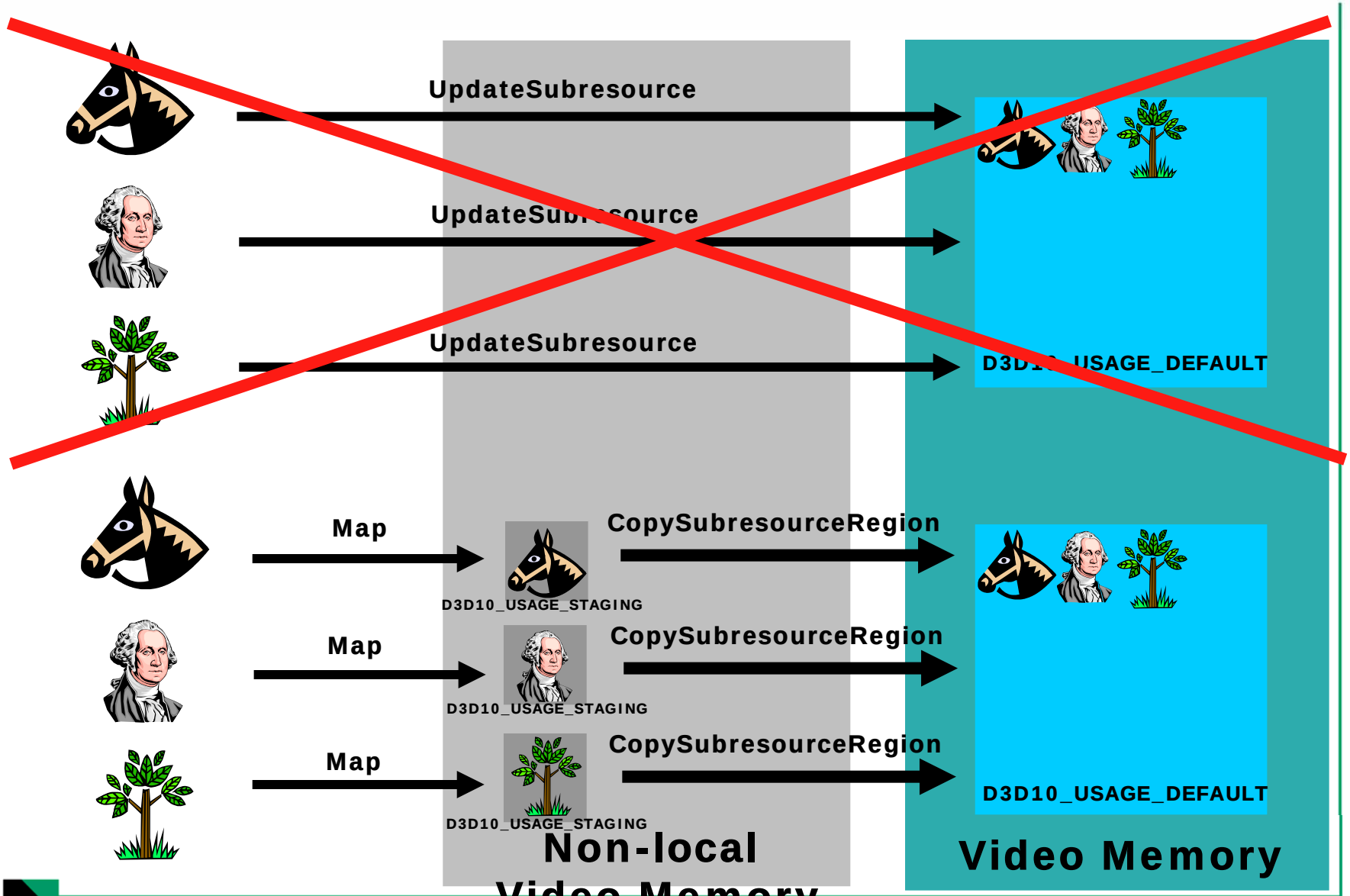
Then upload staging resources into video memory

- **`CopyResource()`**
- **`CopySubresourceRegion()`**

Resource Updates: Textures (2)



Resource Updates: Textures (3)



Resource Updates: Buffers

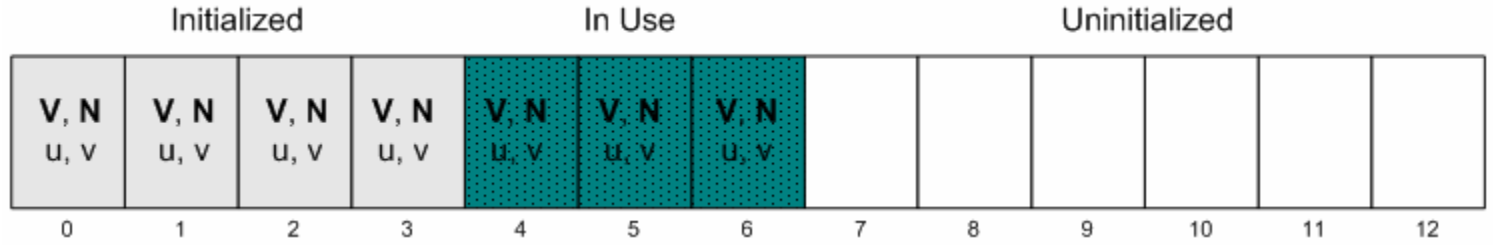
AMD Smarter Choice

To update a Constant buffer

- Map(D3D10_MAP_WRITE_DISCARD, …);
- UpdateSubResource()

To update a dynamic Vertex/Index buffer

- **`Map(D3D10_MAP_WRITE_NO_OVERWRITE, …);`**
- Ring-buffer type; only write to empty portions of buffer
  - **`Map(D3D10_MAP_DISCARD)`** when buffer full
- UpdateSubresource() not as good as Map() in this case



DX10 Batch Performance

AMD
Smarter Choice

The truth about DX10 batch performance

“Simple” porting job will not yield expected performance

Need to use DX10 features to yield gains:

- Geometry instancing
- Intelligent usage of state objects
- Intelligent usage of constant buffers
- Texture arrays

Geometry Shader Recommendations

AMD
Smarter Choice

GS can write data out to memory

- This gives you “free” ALUs because of latency

Minimize the size of your output vertex structure

- Yields higher throughput
- Allows more output vertices (max output is 1024 floats)
- Consider adding pixel shader work to reduce output size

Geometry Shader Recommendations 2

AMD
Smarter Choice

Cull triangles in Geometry Shader

- Backface culling
- Frustum culling

Minimize the size of your input vertex structure

- Combine inputs into single 4D iterator (e.g. 2 UV pairs)
- Pack data and use ALU instructions to unpack it
- Calculate values instead of storing them (e.g. binormal)

Color Clears

AMD
Smarter Choice

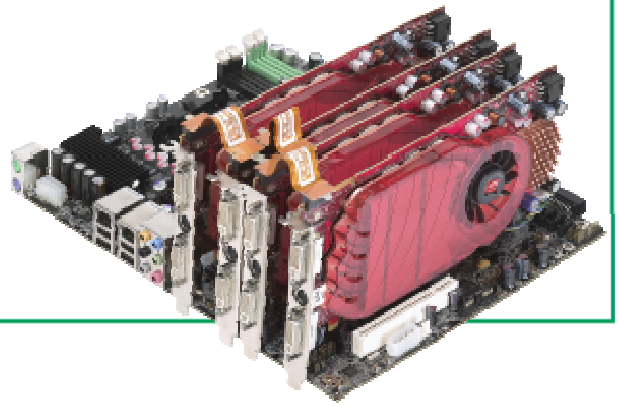
Clearing of color render targets is ***not*** free on R6x0

- Cost is proportional to number of pixels to clear
- The less pixels to clear the better!

Here the rule about minimum work applies:

- Only clear render targets that need to be cleared!
- Exception for MSAA RTs that need clearing every frame
- Could even use partial Clears using draw calls

RT clears are *not* required for optimal multi-GPU usage



Shader Compiler – Parallel Code

AMD Smarter Choice

R6x0 unified shader operates on 5 scalars per instruction

- x,y,z,w scalar units for generic MAD-style operations
- t unit for MAD and special operations (“transcendental”)

Write parallel shader code for best performance

- Enable optimal instruction throughput
- Avoid scalar dependencies

Execute scalar instructions first when mixed with vector ops

- Use parentheses can help

```
float  a;
float  b;
float4 V;

V =  V * (a * b);
```

Shader Compiler – Transcendental Unit

AMD Smarter Choice

Operations handled by t unit (on top of generic MAD ones)

- Integer multiplication and division
- Bit shifts
- Type conversion (float to int, int to float)
- Division, reciprocal, square root, reciprocal square root
- log, exp, pow
- sin, cos

Too many of those can cause a bottleneck in your code

In particular watch out for type conversions

- Remember to declare constants in the same format as the other operands they're used with!

Shader Compiler – ALU:TEX ratio

AMD
Smarter Choice

ALU:TEX ratio on R600/RV670 series is **4:1**

- This is the basic ratio for <=64 bits formats with no filtering
- This ratio is for *hardware* instructions, not D3D ASM ones!
- Feed your shaders into GPUShaderAnalyzer for analysis!

The following require more samples thus affect the ideal ratio

- Advanced filtering (trilinear, anisotropic)
- 128 bit texture formats and 64-bit *integer* format
- Volume textures
- Special texture instructions (e.g. SampleGrad)

Example: sampling three 16161616f textures with trilinear requires up to 24 ALU instructions for a balanced shader

- 3 (textures) x 2 (trilinear) x 4 (ALU ratio) = 24

Shader Compiler – Indexing

AMD
Smarter Choice

Indexed temporaries will reduce performance when stored in external video memory

- Avoid declaring arrays of temporaries
- The performance impact of temp indexing can be dramatic!

Indexed constants may also be subject to direct memory access

- Especially when indexing larger arrays

You can detect such cases with GPUShaderAnalyzer

- Watch out for the following instructions:

  MEM_SCRATCH_WRITE

  MEM_SCRATCH_READ

  VFETCH

Alpha Test

AMD
Smarter Choice

Alpha test deprecated in DirectX 10

- Use **`discard()`** or **`clip()`** in HLSL pixel shaders

This requires the creation of two shader versions in DX10!

- One without **`clip()`** for opaque primitives
- One with **`clip()`** for transparent primitives

Don't be tempted to use a single **`clip()`**-enabled shader

- This will impact GPU performance w.r.t. Z culling
- A single shader with a static/dynamic branch will still not be as good as having two versions

Side-effect: contribution towards "shaders explosion"

Put **`clip()`** / **`discard()`** as early as possible in shader

- Compiler may be able to skip remaining instructions

Accessing Depth and Stencil

AMD Smarter Choice

DX10 enables the depth buffer to be read back as a texture

Enables features without requiring a separate depth render

- Atmosphere pass, soft particles, DOF, forward shadow mapping, screen-space ambient occlusion, etc.

This is proving very popular to most game engines

- DX10.1: no problem, just re-use the depth buffer in all cases
- DX10: requires a separate depth render path for MSAA
  - Store depth in alpha of main FP16 RT
  - Render depth into texture in a depth pre-pass
  - Use a secondary render target in main color pass

Shadow Maps

AMD Smarter Choice

Avoid using **DXGI_FORMAT_D24_UNORM_S8_UINT** for depth shadow maps

- Reading back a 24-bit format is a slow path on R6x0
- No need for stencil for shadow maps anyway

Recommended depth shadow map formats:

DXGI_FORMAT_D16_UNORM

- Fastest shadow map format
- Precision is enough in most situations
  - Just need to set your projection matrix optimally

DXGI_FORMAT_D32_FLOAT

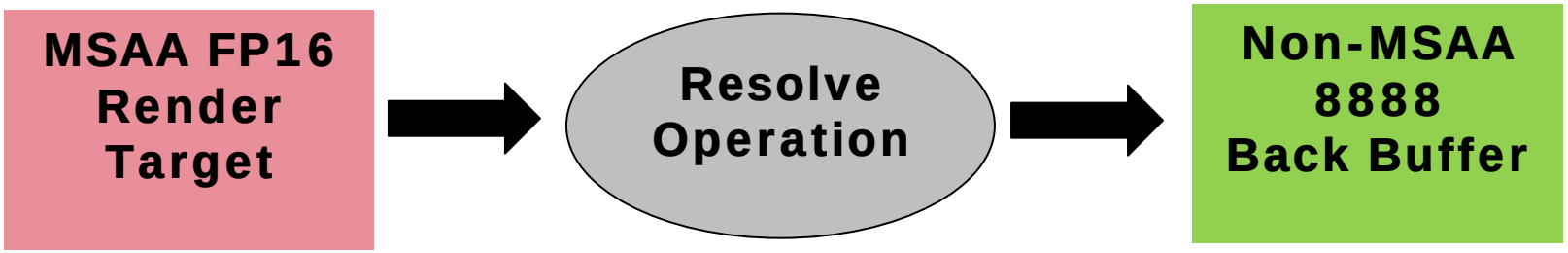
- High-precision but slower than the 16-bit format

Multisampling Anti-Aliasing

AMD Smarter Choice

Remember to NOT create your back buffer as multisampled!

- In most cases all multisampled rendering occurs on RT
  - Typically FP16 render target for HDR rendering
- Back buffer only receives resolved contents of RT



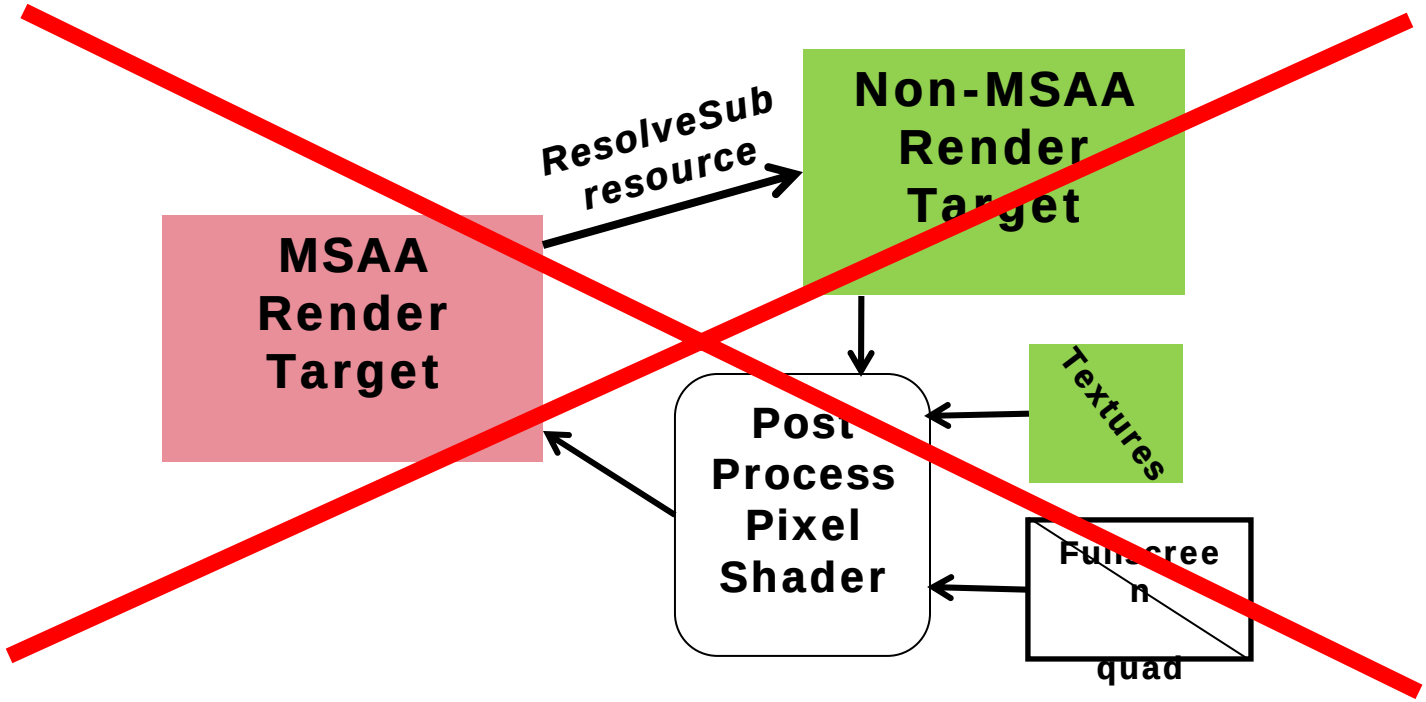
MSAA resolve operations are not free (on any hardware)

- This means **`ResolveSubresource()`** costs performance
- It is *essential* to limit the number of MSAA resolves
- Requires good design of effects and post-process chain

Reducing the number of MSAA Resolves 1

AMD Smarter Choice

Avoid post-processing image ops on the MSAA buffer!

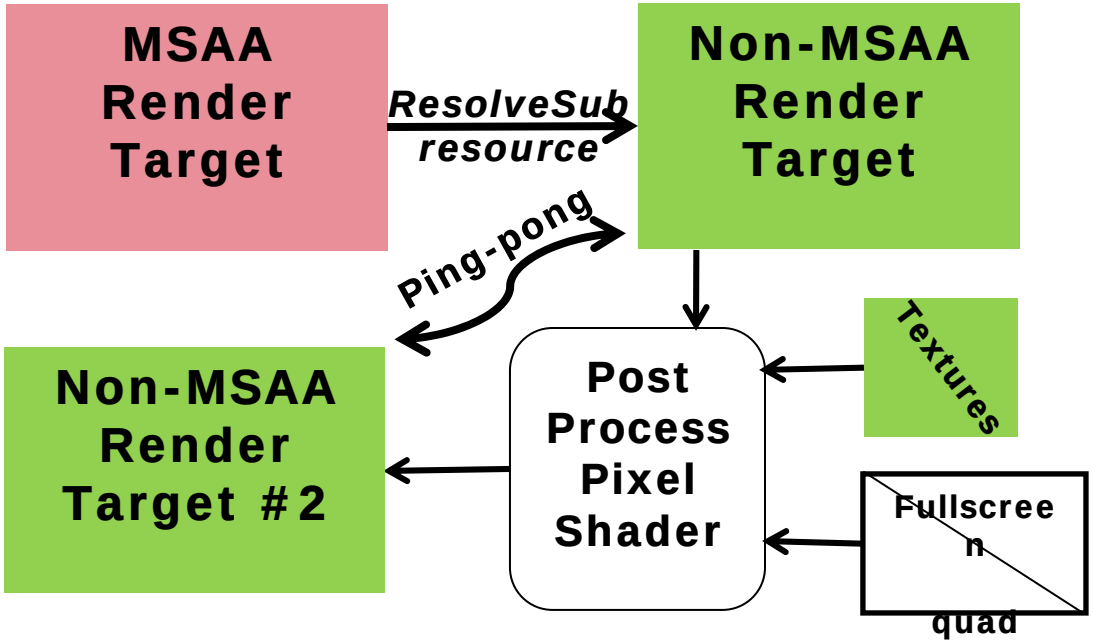


Especially bad with multiple post-process operations
- Requires multiple resolves
- MSAA rendering for no gain (no edges to antialias!)

Reducing the number of MSAA Resolves 2

AMD Smarter Choice

Post-processing image ops should be done on resolved buffer



Resolve MSAA render target once, ***then*** apply post-processing

Ping-pong the two non-MSAA RTs for multiple effects

Reducing the number of MSAA resolves 3

AMD Smarter Choice

`Load()` gives individual access to MSAA samples in pixel shader

Allows custom pixel shader-based resolves

- HDR-correct resolves
- Deferred Shading resolves
- Shadow map resolves (DX10.1)

This also enables “cheap” resolves to be carried out

- Can be needed for some post-process effects
- Load() and process a single sample
- Single-sample resolves faster than fixed-function

In the end the scene needs only a ***single*** “real” MSAA resolve

- Subsequent resolves need only be single sample
- ...or better yet, not required

Post-Tone Mapping HDR MSAA Resolve

Standard MSAA resolve occurs before tone-mapping

HDR MSAA resolve should be done ***after*** tone-mapping!

$$ToneMap\left(\frac{\sum_{n=1}^{\#samples} Sample(n)}{\#samples}\right) != \frac{\sum_{n=1}^{\#samples} ToneMap\left(Sample(n)\right)}{\#samples}$$

Standard resolve

Custom resolve

A taste of DX10.1

AMD
Smarter Choice

DX10.1 released with Windows Vista SP1

- Should be a couple of weeks from now

Next Catalyst release adds DX10.1 support to HD3x00 series

DX10.1 features:

- MSAA depth buffer readback as texture
- Cube map arrays
- Per-pixel coverage output mask
- Mandatory HW support (MSAA 4x, FP32 filtering, etc.)
- Per-sample execution in pixel shader
- Separate MRT blend mode
- New 4.1 shader model (Gather4, samplepos, etc.)
- More precision etc.

Conclusion

AMD
Smarter Choice

Ensure the DX10 API is used optimally

- Geometry Instancing!
- Right balance of resource updates

Limit Geometry Shader output size to a minimum

Only Clear() when and where necessary

Watch out for unexpected shader compiler output!

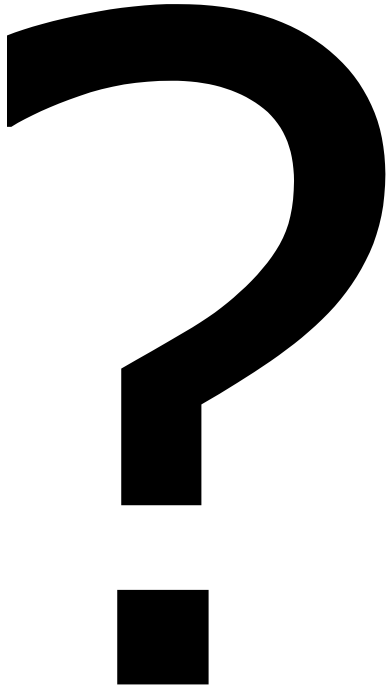
- Use GPUShaderAnalyzer

Using a minimum number of MSAA resolves is *essential*

Create a DX10.1 device if supported and use what you need

Questions?

AMD
Smarter Choice



nicolas.thibieroz@amd.com